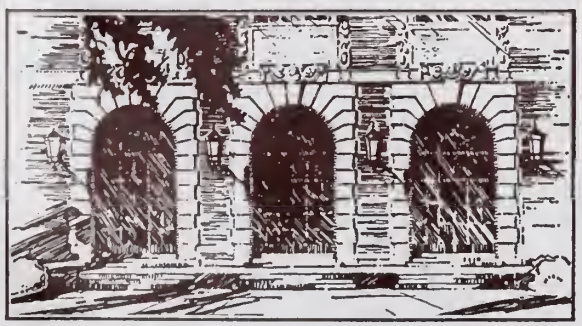

UIUCDCS-R-71-420

THE STRUCTURE AND DESIGN PHILOSOPHY OF OL/2

-AN ARRAY LANGUAGE-

PART II: ALGORITHMS FOR DYNAMIC PARTITIONING

by
J. Richard Phillips

September, 1971

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-71-420

THE STRUCTURE AND DESIGN PHILOSOPHY OF OL/2

-AN ARRAY LANGUAGE-

PART II: ALGORITHMS FOR DYNAMIC PARTITIONING

by

J. Richard Phillips

1971

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

This work was supported in part by the National Science Foundation
Grant No. US NSF-GJ-328.

TABLE OF CONTENTS

PREFACE

Dynamic partitioning is an array language construct which was conceived during the design of the OL/2 language by the author. It is interesting because it allows one to write array algorithms for areas which have previously been difficult or impossible; for instance, the direct methods of linear algebra. However, dynamic partitioning is not restricted to array languages with an algebraic structure like OL/2. The basic ideas and the information structure are applicable to other array languages with different data types and different array operations. The purpose of this paper is to provide a formal description of dynamic partitioning and to document the basic algorithms in a language independent form. This should be valuable to anyone who is concerned with the construction and design array languages.

The first part of this paper will appear in CACM [7], while the latter part, which is concerned with algorithms for implementing partitioning, will be described in this paper in a Knuth-like notation [6].

Dynamic partitioning was first conceived in the latter part of 1969; since then several persons have been instrumental in its implementation. The first was John Gaffney, who made a feasibility test. The second was H. C. Adams, who implemented the present version for OL/2 as a Master's thesis [1]. All of this would have been impossible without

the efforts of Robert Bloemer and Dale Jurich, who have been major contributors to the implementation of OL/2. I would like to express my appreciation to each of these people for their contributions, confidence, and enduring spirit!

ABSTRACT

The classical process of partitioning an array into subarrays is extended to a more useful array language operation. Various modes of partitioning are defined for different types of arrays, so that subarrays may vary over the original array in a nearly arbitrary manner. These definitions are motivated with several realistic examples to illustrate the value of partitioning for array languages.

Of general interest is the data structure for partitioning. This consists of dynamic tree structures which are used to derive and maintain the array control information. These are described in sufficient detail to be of value in the design of other array languages. The description presented in this paper is implemented in a new array language, OL/2, currently under development at the University of Illinois.

KEY WORDS AND PHRASES: dynamic partitioning, array partitioning, array language, data structure, tree structure, programming language design, array control blocks, partition control blocks

CR CATEGORIES: 4.12, 4.2, 4.22

Introduction

Partitioning, in the usual sense, consists of separating an array into subarrays by inserting partition lines between certain rows and columns. The purpose is to implicitly define the subarrays by their relative position in the original array so that they may be renamed or indexed for notational purposes. In this primitive form partitioning is limited, but in a more general setting it becomes a very powerful array operation.

In essence, dynamic partitioning, as defined here and in the OL/2 language [7], allows the partitioning lines to vary over the array. This in turn allows various subarrays to be referenced in array expressions. If we also include the option of selecting subarrays which vary in size and shape, then most direct methods of linear algebra can be implemented in a simple and straightforward manner. It is also interesting to observe that an array language with partitioning can use storage as efficiently as an element language such as PL/1 or ALGOL. The crucial point is not the array language but the algorithm itself. If the algorithm is amenable to array notation and temporary arrays are not required by the array expressions, then the algorithm may be implemented with a properly designed array language which uses the same amount of storage and the same number of operations as an element language. To support this point we present in the appendix two well-known algorithms using the OL/2 language: the Crout decomposition algorithm and the Cholesky method.

The purpose of this paper is to describe the various modes of partitioning and a strategy for implementing it. A detailed description of the implementation for the OL/2 language on the IBM 360/75 appears in [1]. Other array languages usually have some provision for selecting subarrays. In particular, there is the "take" and "drop" functions in APL [5], and the "ranges" and extraction operators ◁ and ▷ in the article by Bayer and Witzgall [2]. The SPEAKEASY language [3] and the MPL language [4] are examples of other array languages which have a simple form of subarray selection.

Basic Data Types

The basic types of arrays for which partitioning is defined are given in Table I. From this basic set a larger class of arrays may be constructed, such as block tridiagonal matrices or band matrices. Even though partitioning applies to these composite arrays, it is sufficient to consider a partitioning strategy based only on the arrays in Table I.

The types of arrays which have been chosen are the common forms that are encountered when solving problems in linear algebra. Since the storage for arrays is usually critical, it becomes important not to store the elements which are theoretically zero. The arrays still operate on each other according to the rules of linear algebra, but the storage and number of operations is reduced to a minimum. As a matter of convenience, the elements of all arrays are stored linearly in row major order. This storage scheme together with the various types of arrays determine, in part, the information that must be inserted into the data structure which is described in subsequent sections.

TABLE I Basic Array Types

ARRAY TYPE τ	VIRTUAL BOUNDS		STORED ELEMENTS $S(\tau)$
	I	J	
STRICTLY LOWER TRIANGULAR	1:N	1:N	$1 \leq J < I \leq N$
LOWER TRIANGULAR	1:N	1:N	$1 \leq J \leq I \leq N$
DIAGONAL	1:N	1:N	$1 \leq I = J \leq N$
STRICTLY UPPER TRIANGULAR	1:N	1:N	$1 \leq I < J \leq N$
UPPER TRIANGULAR	1:N	1:N	$1 \leq I \leq J \leq N$
TRIDIAGONAL	1:N	1:N	$\lvert I - J \rvert \leq 1;$ $1 \leq I, J \leq N$
RECTANGULAR	1:M	1:N	$1 \leq I \leq M;$ $1 \leq J \leq N$

Modes of Partitioning

The simplest form of partitioning is referred to as static partitioning. It is defined by fixing the partitioning lines permanently after specific rows or columns of an array. Figure 1 illustrates this type of partitioning for several of the arrays in Table I.

More precisely, we define partitioning for all of the triangular and diagonal arrays of Table I by having the partitioning line reflect off of an imaginary diagonal line which starts in the upper left corner of the array and proceeds to the lower right corner. Since these arrays are square, by definition, the effect is to simultaneously partition the array after the same row and column.

Rectangular arrays, on the other hand, are partitioned in the usual way. That is, they are partitioned after any row or column of the array with the row and column partitioning lines being independent of each other.

Another mode of partitioning, which is referred to as diagonal partitioning, is also defined. To simplify matters assume that the array to be partitioned is a square matrix of order n. Then one may specify any part of the matrix which is compatible with one of the triangular or diagonal arrays in Table I; for example, one may specify the strictly upper triangular part of the matrix. In this instance, the effect is equivalent to placing a diagonal partitioning line just above the principal diagonal of the matrix.

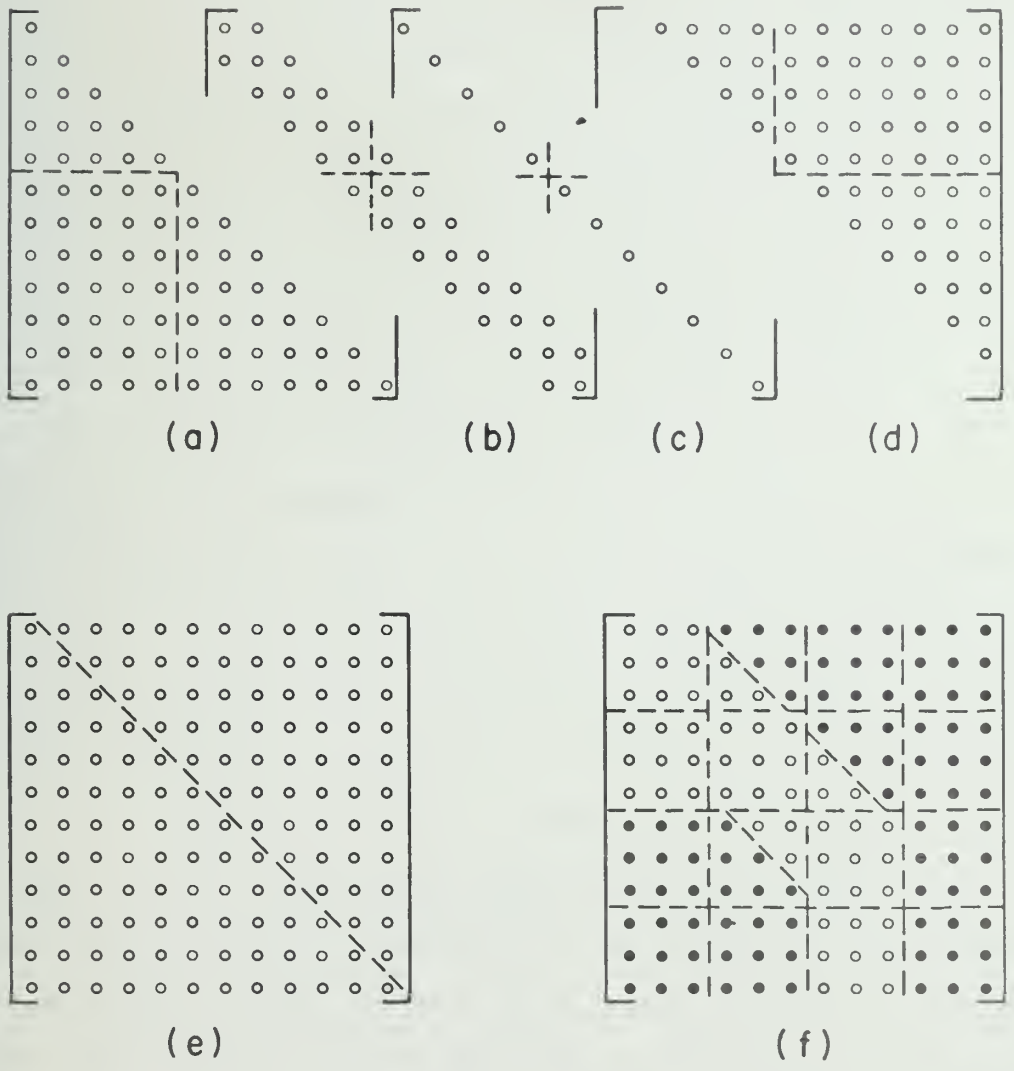

FIG. 1. Elementary partitioning of the basic types of arrays: (a) lower triangular; (b) tridiagonal; (c) diagonal; (d) strictly upper triangular; (e) diagonal partitioning; (f) a combination of rectangular and diagonal partitioning with a selection of a subset of elements represented by the solid marks.

In all cases of partitioning, no additional storage is needed. Partitioning merely allows one to select subarrays of various sizes and shapes. These subarrays are then available for referencing in array expressions, so that array algorithms may be expressed in a simple and direct manner.

We may extend the partitioning definitions to the dynamic realm by allowing the partitioning lines to be defined by expressions. Thus, in dynamic partitioning the subarrays vary over the parent array in an arbitrary manner as the values of the expressions change. At the same time, the subarrays remain uniquely defined by their relative position in the parent array, so that their name remains unchanged in any array expression.

To illustrate, consider the following statements which are extracted from one of the examples in the appendix.

```
LET A BE A MATRIX OF ORDER (N); FOR K = 1,2,...,N;
PARTITION A AFTER ROW K-1 AND AFTER COLUMNS K-1, K;
SET C = A<2,1>, X = A<1,2>, Y = A<2,2>, B = A<1,3>,
AND M = A<2,3>;
```

The matrix A is partitioned into six subarrays in Figure 8a by the PARTITION statement, and five of the subarrays are renamed in a corresponding SET statement. The subarrays of A are specifie with the aid of special parentheses < > so that the common parentheses can be reserved for specifying the elements of the matrix. Thus, using conventional indexing, A<1,1> refers to the subarray

in the upper left corner, and A<2,1> to the subarray immediately below A<1,1>.

In the above example the partitioning is dynamic, since the FOR statement alters the expressions which define the position of the partitioning lines. As the value of K is altered, the subarrays C, X, Y, and B move over the matrix A in a uniform manner. The meaning of the statements above for the cases K = 1 and K = N are discussed in the last section, which is concerned with the definition of null operands.

We now extend the operation of partitioning. Since the partitioning lines implicitly define the subarrays, each subarray is a candidate for further partitioning. In general, partitioning may be applied to all subarrays which result from previous partitionings, whether they have been explicitly specified in a SET statement or not. The process is referred to as nested partitioning.

One of the advantages of nested partitioning is that overlapped subarrays of various sizes and shapes may be referenced. That is, the programmer has equal access to any of the arrays which have resulted from nested partitioning, provided only that the reference is within the scope of the PARTITION statement. However, it is interesting to note that in practice most algorithms require only a few levels of partitioning. For instance, the algorithms of Cholesky and Crout, in the appendix, require only one and two levels of nested partitioning respectively.

Finally, there is an even more general set of subarrays that may be selected by overlaying several independent partitionings. This is achieved by first defining two arrays or subarrays to be

equivalent; for instance, SET T = A. This means that both T and A refer to the same set of elements. Then, by allowing T and A to be partitioned independently of each other, we obtain independent sets of subarrays.

In conclusion, we have two basic types of partitioning: static and dynamic. With either of these types we may have nested partitioning to an arbitrary level. At each level, there is a mode of partitioning corresponding to a classification of array types. That is, one mode for triangular and diagonal arrays, another mode for rectangular arrays which is the classical rectangular partitioning, and finally a diagonal partitioning mode which selects triangular or diagonal subarrays. Complementing these options is the ability to define one or more independent partitions on the same array by overlaying the partitions.

The Data Structure for Partitioning

Partitioning is implemented by constructing a general tree structure T. The root of the tree T corresponds to some array A which is explicitly declared by a LET statement, while the subarrays of A form the roots of the subtrees of T at the next level. If any of the subarrays are partitioned, then nodes are added to the subtree at the next level. Therefore, in general, the entire data structure consists of independent tree structures: one for each explicitly declared array.

In practice, it is not necessary to build the entire tree structure; in fact, it is preferable to build only those parts which are going to be referenced by the programmer. This can be accomplished if an explicit specification, such as the SET

statement in the last section, is required with each partition statement.

The justification for this convention is based on several observations. First, in many algorithms the programmer needs to reference only a few of the subarrays which result from partitioning, and secondly, the size of the tree structure can grow very rapidly if all of the subarrays are represented by nodes in the tree. An additional advantage, which perhaps is equally important from a semantic point of view, is that the SET statement helps to document the algorithm in a natural way.

In some cases, however, it is convenient to reference subarrays which have not been explicitly specified in a SET statement. In this case, the tree structure must be altered dynamically during program execution. The partitioning lines still determine the number of subarrays and their relative position in the parent array, but it is no longer possible to determine, a priori, which subarray is going to be referenced. For example, if A is the parent array, then A<I,J> is a reference to a subarray that is not known during compilation. However, it also follows that the size and shape of the subarray A<I,J> may now change as the values of the subscripts change. This provides a rather powerful form of referencing subarrays when combined with dynamic partitioning.

It is natural to refer to the tree structures which must be altered at execution time as dynamic, and those which remain structurally invariant as static. Whether a tree structure is dynamic or static is independent of whether or not the partitioning is dynamic or static. In fact, most dynamic partitioning can be

realized with a static tree structure.

In order to illustrate some of the ideas discussed so far, we construct an example using several levels of static partitioning. Figure 2a shows the partitioning of the array A which corresponds to the statements below, while Figure 2b shows a representation of the tree structure.

```
LET A BE A LOWER TRIANGULAR MATRIX OF ORDER (15);
PARTITION A AFTER ROWS 6, 9; SET B = A<2,1>, C = A<2,2>,
AND D = A<3,2>; SET U TO THE STRICTLY UPPER TRIANGULAR
PART OF A<3,1>;
  PARTITION A<3,1> AFTER ROW 1 AND COLUMN 1; SET L TO
THE LOWER TRIANGULAR PART OF A<3,1><2,2>, E TO THE
DIAGONAL PART OF A<3,1><2,2>;
  PARTITION E AFTER ROW 2; SET F TO E<2,2>;
```

The first partition statement divides the original array A into six subarrays which can be referenced. The subarrays are subscripted as though A were square and not lower triangular. Therefore, A<1,2>, A<1,3>, and A<2,3> are undefined and cannot be referenced. This subscripting convention applies to all of the arrays in Table I. Also observe that the subarrays on the diagonal inherit the lower triangular attribute from the parent array A.

The subarray A<3,1> is not specified in a SET statement; however, its subscripts are constants so it is equivalent to an explicit specification. For this reason the corresponding tree structure is static.

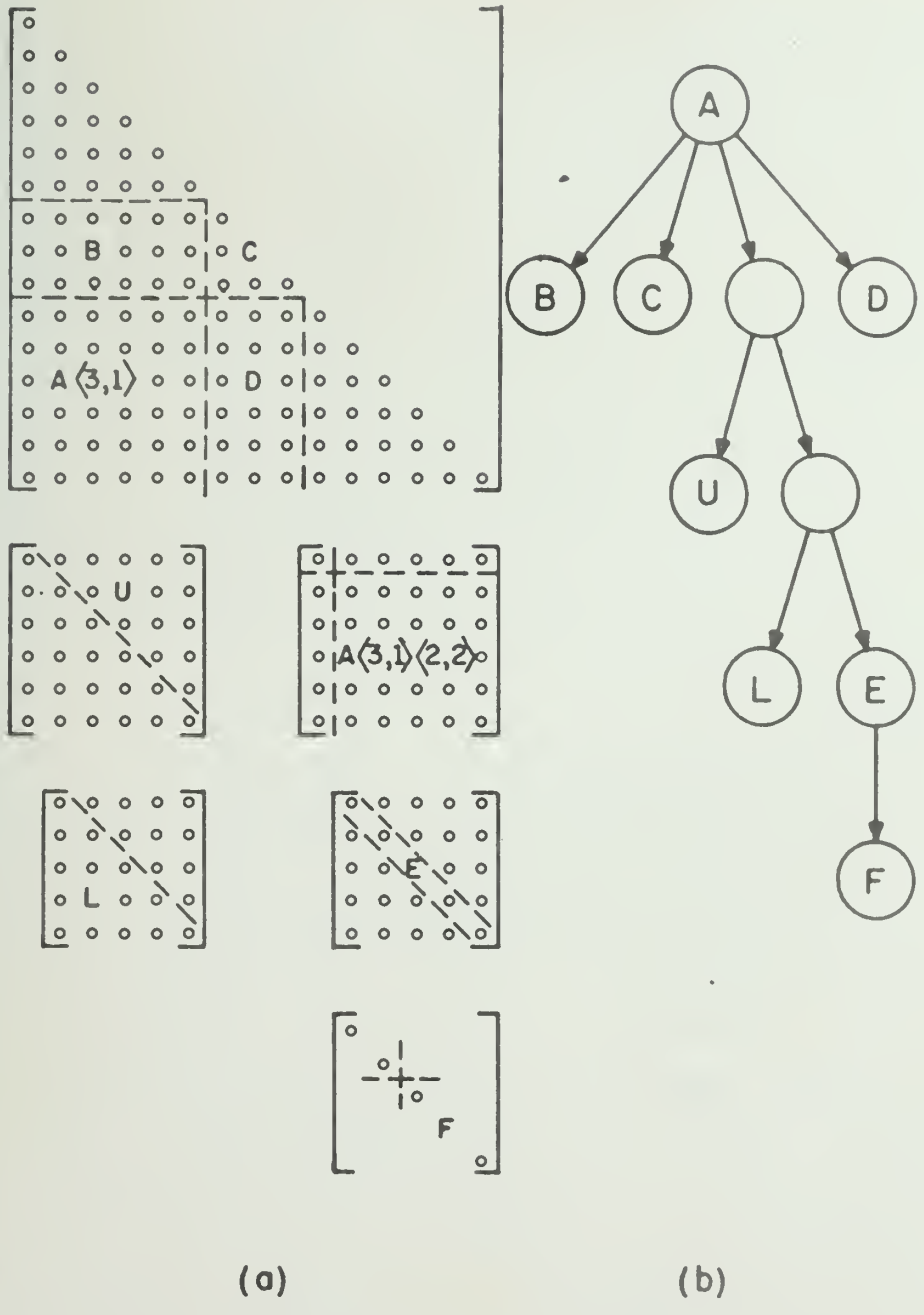

FIG. 2. Multilevel partitioning of a lower triangular array: (a) array and its partitioned subarrays; (b) corresponding tree structure. The nodes for A<3,1> and A<3,1><2,2> are not labeled since they are not explicitly renamed.

The notation A<3,1><2,2> in the second SET statement specifies the subarray whose position is defined by the subscripts <2,2> in the parent array A<3,1>. This is illustrated in Figure 2a.

Array Control Block

Each node of a tree structure is referred to as an array control block (ACB). The purpose of this control block is to hold control information for accessing the elements of an array. In general, we are interested in accessing all elements of a prescribe array or subarray; we consider this to be an intrinsic property of array languages.

Figure 3 shows an ACB node with its fields, while Table II lists some of their characteristics. In general, the size of the ACB node depends upon the dimension of the array. Since we have defined partitioning only for one and two dimensional arrays we shall limit our discussion to these cases. However, it should be noted that higher dimensional arrays can be represented by a simila ACB structure, and that the corresponding tree structures consist of only the root nodes. Furthermore, vectors (column or row) are special cases of rectangular arrays, and need not be distinguished from more general rectangular arrays, except by the value of the dimension field. The same argument may be applied to scalars which result from partitioning, since they are also special cases of rectangular arrays. A scalar which is not simultaneously an

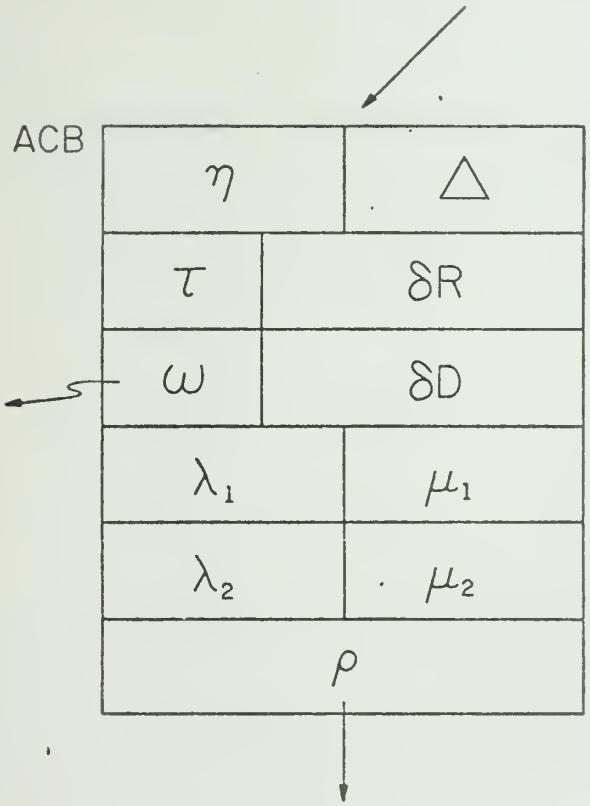

FIG. 3. Array control block (ACB) node.

TABLE II Array Control Block (ACB) Fields

SYMBOL	FIELD	ATTRIBUTE
η	Name	Character String
Δ	Dimension	Integer
τ	Type	Integer
δD	Diagonal Increment	Integer
δR	Row Increment	Integer
ω	Origin	Pointer
λ_i	Lower Bounds	Integer
μ_i	Upper Bounds	Integer
ρ	Partition	Pointer

element of an array is not described by an ACB, but is treated in a normal manner. Therefore, we focus our attention upon two dimensional arrays only.

The root node is formed from information that is contained in a LET statement; in fact, we may set up a one to one correspondence between the explicitly declared arrays in the LET statement and the root nodes of the tree structures. To simplify the discussion, we shall assume that the root nodes are formed upon block entry and are destroyed, along with their subtrees, only upon block exit. This condition will be relaxed later.

Most of the information for filling a root node is easily obtained. For instance, the name (η), dimension (Δ), type (τ), lower bounds (λ_i), and upper bounds (μ_i) are explicitly specified in a LET statement, or are implicitly specified as in the case of the type field where a default of rectangular is assumed. The partition pointer ρ, which is not used until partitioning is attempted, is set to the null pointer Λ.

In order to define the origin, ω, we adopt the convention that $\lambda_i=1$ for each dimension and for all arrays and subarrays. Then we define ω as the location of the first accessible element of the array described by the ACB node. That is, beginning a labeling of any two dimensional array or any subarray with row 1 and column 1, we let ω point to the element in position (1,1) unless its type is strictly lower triangular or strictly upper triangular, in which case we let it point to the element in position (2,1) or (1,2) respectively.

TABLE III Initial Values for δD and δR in Root Nodes

ARRAY TYPE	SYMBOL	δD	δR
Strictly Lower Triangular	(SLT)	1	0
Lower Triangular	(LT)	1	1
Diagonal	(D)	0	0
Strictly Upper Triangular	(SUT)	-1	$\mu_2-\lambda_2-1$
Upper Triangular	(UT)	-1	$\mu_2-\lambda_2$
Tridiagonal	(TD)	0	2
Rectangular	(R)	0	$\mu_2-\lambda_2+1$

Finally, for the root nodes only we define the ACB fields δR and δD from the entries in Table III. We refer to ω, δR, and δD as control fields, and we consider their derivation in the next section.

The Control Fields δR, δD, and ω

The object of this section is to show that the elements of any array or subarray may be accessed efficiently by using the information in the ACB nodes. We shall distinguish two cases: access of contiguous elements and access of an arbitrary element. For the contiguous case, the row increment, δR, and the diagonal increment, δD, are used as initial values.

We define δD in Table III as the number of elements which can be accessed in row i+1 minus the number of elements which can be accessed in row i. From Table I it follows that δD is constant for all i, and can be determined at compile time from the array type.

In order to define δR, we introduce the following notation. Let $S(\tau)$ be the set of subscript pairs (i,j) which are defined by the relations in the last column of Table I, where τ is the array type. Then $(i,j) \varepsilon S(\tau)$ means that the array element in position (i,j) is defined and can be accessed. If $(i,j) \varepsilon S(\tau)$ for some array (or subarray), then the location of this element is denoted by L(i,j).

Under the conventional row major order storage scheme and for root nodes only, we define δR in Table III by

$$\delta R = L(2,j) - L(1,j) \qquad (2,j), (1,j) \in S(\tau)$$

for all τ, except diagonal and strictly lower triangular, which are defined as zero. It is easy to see from Table III that δR can be constructed from the information that can be extracted from a LET statement.

As indicated previously, ω, δR, and δD are constructed so as to access successive elements of an array. It is obvious that the next element in row i can be accessed efficiently by means of

$$L(i,j+1) = L(i,j) + 1 \qquad (i,j), (i,j+1) \in S(\tau) \qquad (1)$$

for all τ; this relation also holds for any subarray, since the row elements in any subarray must necessarily be continguous. Diagonal arrays, of course, do not have a next row element so (1) is never satisfied.

For accessing successive elements in any column, let

$$\delta R(1) = \delta R, \quad \delta R(i+1) = \delta R(i) + \delta D \qquad (2)$$

for all ACB nodes. Then, the relation for the location of the next element in column j is defined by

$$L(i+1,j) = L(i,j) + \delta R(i) \qquad (i+1,j), (i,j) \in S(\tau) \qquad (3)$$

for all τ.

Formulas (2) and (3) are valid not only for arrays described by root nodes, but for all of their subarrays as well. The verification for arrays described by root nodes follows from the values of δR and δD given in Table III, while the verification for subarrays requires a proof which we now present.

Let A_s be a subarray of A_p such that the $node_p$ is the parent of $node_s$. By definition, we say that A_s is a subarray of A_p if and only if every accessible element of A_s, considered as a set, is also an accessible element of A_p, in other words $A_s \subseteq A_p$. This excludes, for example, A_s being the tridiagonal part of an upper triangular A_p because the elements on the lower diagonal of A_s are not contained in A_p.

We use induction on the tree level, where $node_p$ is at level k and $node_s$ is at level k+1. The case k=1, the root node level, follows from Table III. Thus, assume that (2) and (3) are valid for $node_p$. Then, it follows that

$$\delta R_s = \delta R_p + (i-1)\delta D_p \tag{4}$$

and

$$\delta D_s = \delta D_p \tag{5}$$

where i is the i-th row of A_p and simultaneously the first row of A_s. By definition of a subarray, i always exists and is equal to or greater than unity. From (2) and (4) it follows that

$$\delta R_p(i) = \delta R_p + (i-1)\delta D_p = \delta R_s \tag{6}$$

Now choose an arbitrary element of A_s and denote its subscripts by (m,j). We will show that (3) is valid for A_s provided that (m,j) and (m+1,j) ε $S_s(\tau)$, where $S_s(\tau)$ corresponds to the set of subarray elements of A_s which can be accessed. Since the arbitrary element of A_s is contained in A_p, its subscripts with respect to A_p are (i+m,j') ε $S_p(\tau)$ for some j'. Therefore, since (3) and (2) are valid for $node_p$ it follows from (6), (5), and (2) t

$$L_s(m+1,j) = L_p(i+m+1,j') = L_p(i+m,j') + \delta R_p(i+m)$$

$$= L_s(m,j) + \delta R_p(i) + (m-1)\delta D_p$$

$$= L_s(m,j) + \delta R_s + (m-1)\delta D_s$$

$$= L_s(m,j) + \delta R_s(m).$$

Since the element in A_s was arbitrary, (3) is valid for accessing any consecutive column elements of A_s, and this completes the proof. This result is summarized in Theorem 1.

We note that (4) and (5) define the rules for constructing the quantities δR and δD for any subtree node. The only remaining unknown is the value of i, but this is obtained from the partitioning information that is discussed in the next section.

So far we have shown that successive elements in any row or any column can be accessed by using the information in an ACB node, along with (1) or (3). Many other alternatives also exist. For example, the successive elements along a diagonal are

accessed by using

$$L(i+1,j+1) = L(i,j) + \delta R(i) + 1 \tag{7}$$

provided $(i+1,j+1)$ and $(i,j) \in S(\tau)$.

Next we consider how the origin of a subarray is determined from information contained in the parent node. As before we shall qualify the arrays and fields by s and p.

Depending upon the array type, we have by definition

$$\omega_p = L_p(i,j) \tag{8}$$

where $(i,j) \in \{(1,1),(1,2),(2,1)\}$. Since A_s is a subarray of A_p, there exist integers $\delta i \geq 0$ and $\delta j \geq 0$ such that

$$\omega_s = L_p(i+\delta i, j+\delta j). \tag{9}$$

Therefore, it follows from (1) and (3) that

$$\begin{aligned} L_p(i+\delta i,j+\delta j) &= L_p(i+\delta i,j) + \delta j \\ &= L_p(i,j) + \sum_{k=i}^{i+\delta i-1} \delta R_p(k) + \delta j. \end{aligned}$$

From the definition of $\delta R(i)$ it follows that for any node

$$\sum_{k=i}^{i+\delta i-1} \delta R(k) = \delta R(i)\delta i + \delta D(\delta i(\delta i-1))/2 \tag{10}$$

where i=1 or 2, depending upon the type τ. Therefore,

$$\omega_s = \omega_p + \delta R_p(i)\delta i + \delta D_p(\delta i(\delta i-1))/2 + \delta j \tag{11}$$

for all τ.

The unknown quanities δi and δj are determined from the partitioning information which is discussed in the next section. Since by definition, i = 1 or 2, we may replace $\delta R(i)$ with either $\delta R(1) = \delta R$ or $\delta R(2) = \delta R + \delta D$. Therefore (8) shows that the origin of any subarray ω_s, can be determined from the information in the parent node and from appropriate partitioning information. We summarize these results in the following theorems:

Theorem 1. (Contiguous Access) If the fields δR and δD of a root node of a tree structure are filled according to Table III, and if the fields of each subtree node are derived from the parent node by

$$\delta R_s = \delta R_p + (i-1)\delta D_p \tag{12}$$

$$\delta D_s = \delta D_p \tag{13}$$

where i is the i-th row of the array described by $node_p$ and also the 1-st row of the array described by $node_s$, then we may locate the successor of any array element described by a node in any row, column, or diagonal by applying respectively

$$L(i,j+1) = L(i,j) + 1, \qquad (i,j+1),(i,j) \in S(\tau) \tag{14}$$

$$L(i+1,j) = L(i,j) + \delta R(i), \qquad (i+1,j),(i,j) \in S(\tau) \tag{15}$$

$$L(i+1,j+1) = L(i,j) + \delta R(i) + 1 \qquad (i+1,j+1),(i,j) \in S(\tau)$$

where τ is any array type and R(i) is defined by (2).

Theorem 2. (Arbïtrary Access) Let the conditions of Theorem 1 be satisfied, and let L(i,j) be the location of an arbitrary element in an array described by a node. Then, for any $\delta i \geq 0$ and $\delta j \geq 0$ such that $(i+\delta i, j+\delta j) \in S(\tau)$,

$$L(i+\delta i,j+\delta j) = L(i,j) + \delta R(i)\delta i + \delta D(\delta i(\delta i-1))/2 + \delta j \qquad (17)$$

for all array types τ, where $\delta R(i)$ is defined by (2).

It follows from (17) that

$$L(i+\delta i,j+\delta j) = \omega + \delta R(i)\delta i + \delta D(\delta i(\delta i-1))/2 + \delta j \qquad (18)$$

where $\omega = L(i,j)$ and $(i,j) \in \{(1,1), (1,2), (2,1)\}$ depending upon the type τ. Therefore, an arbitrary element of any array can be located by using the control fields ω, δR, δD along with the displacements δi and δj which can be determined from the subscripts of a referenced element, say A(k,m). (The round parentheses refer to an element of an array, whereas A<k,m> refer to a subarray of A.) It also follows that (11) is a special case of (17).

Partition Control Blocks

If an array is described by an ACB and is not partitioned, then the partition field ρ is set to Λ, the null pointer, and the ACB node becomes a terminal node in the tree. On the other hand, if it is partitioned, then ρ is set to point to a partition control block (PCB) which contains the necessary information for deriving the nodes for the next level in the tree structure. Structurally, the PCB may be considered as a field in the ACB with a variable number of subfields; however, we prefer to treat

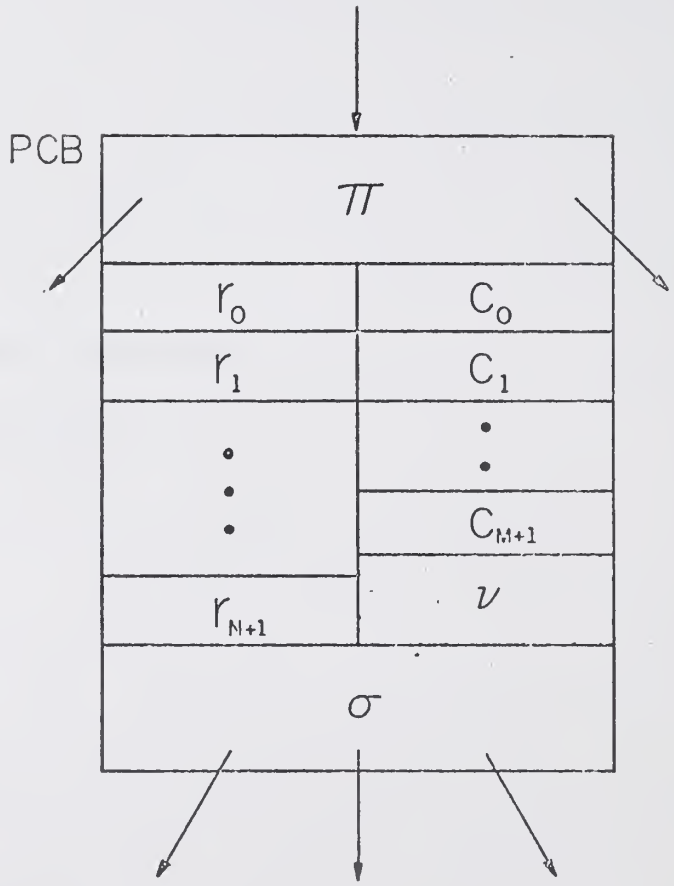

FIG. 4. Partition control block (PCB).

it as a separate control block.

The general structure of the PCB is shown in Figure 4. The field ν contains the number of partition lines defined by the partition statement, namely M+N, while the fields r_o, c_o and r_{N+1}, c_{M+1} contain the lower bounds and upper bounds, respectively, of the array being partitioned.

The partitioning lines, as indicated earlier, are defined by expressions in the partition statement. Corresponding to each row expression R_i and each column expression C_j, there is a row partition field r_i and a column partition field c_j in the PCB. Implicit in the number of partition expressions is also the maximum number of subarrays resulting from the partitioning of the array. Therefore, at compile time one has the information that is necessary for constructing all of the subtree nodes corresponding to a given partition statement. However, under the assumption that only a few of these subarrays will be referenced by the user, we defer the building of the subtree nodes until directed to do so by an explicit specification, such as "SET C=A<2,1>."

The subarray pointer list, σ, in the PCB is used to hold the pointers which point to the subtree nodes at the next level of the tree structure. We can now visualize the basic steps in deriving the information for a subtree node. For example, suppose that the subarray A<2,1> is specified in a SET statement and a PCB for A exists. Then the subarray subscripts <2,1> are used to uniquely reference a pointer in σ, which points to the ACB node for A<2,1>. Next, the subscripts <2,1> are used to uniquely reference the partition lines r_1, r_2, and c_0, c_1 in the PCB,

which define the boundaries of A<2,1>. With this information along with that from the ACB node for A we can fill the ACB node for A<2,1>. A more precise description will be presented in the section on basic algorithms.

The pointer list π in the PCB is used to point to the ACB subtree nodes which describe some "part of" an array, such as the upper triangular part of A. For simplicity, we take π to be an array of six pointers; one for each of the nonrectangular types in Table I. We use the notation $\pi(\tau)$ to refer to the pointer field which corresponds to an array of type τ. If a particular part of an array is not desired, then the corresponding pointer $\pi(\tau)$ is set to Λ. If diagonal partitioning is not desired, then the pointer list π need not exist. On the other hand, if only diagonal partitioning is desired, then the row partition fields r_i, column partition fields c_j, and the pointer list σ need not exist.

The previous discussion illustrates the compactness of the partitioning information in a PCB, and it also illustrates the general technique for adding subtree nodes. Furthermore, it should be clear that the PCB contains sufficient partitioning information for dynamically adding a node to the tree structure at execution time. This happens, for instance, if a subarray reference of the form A<I,J> is encountered, where I and J are variables. In this case, separate ACB nodes are created for each distinct subscript pair <I,J> as they are needed. It is, of course, possible to use a

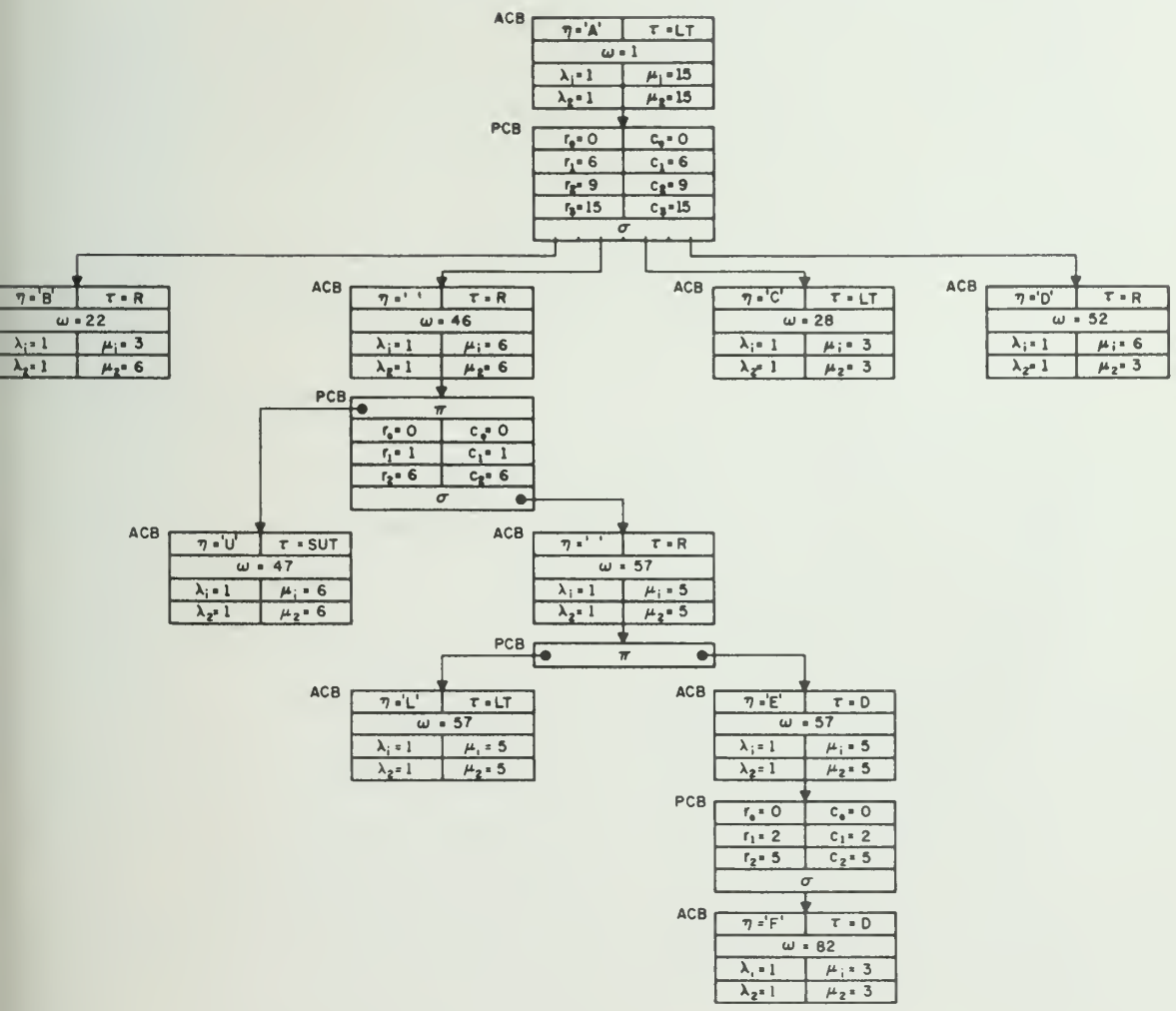

FIG. 5. Tree structure with array control blocks (ACB) and partition control blocks (PCB) corresponding to Figure 2.

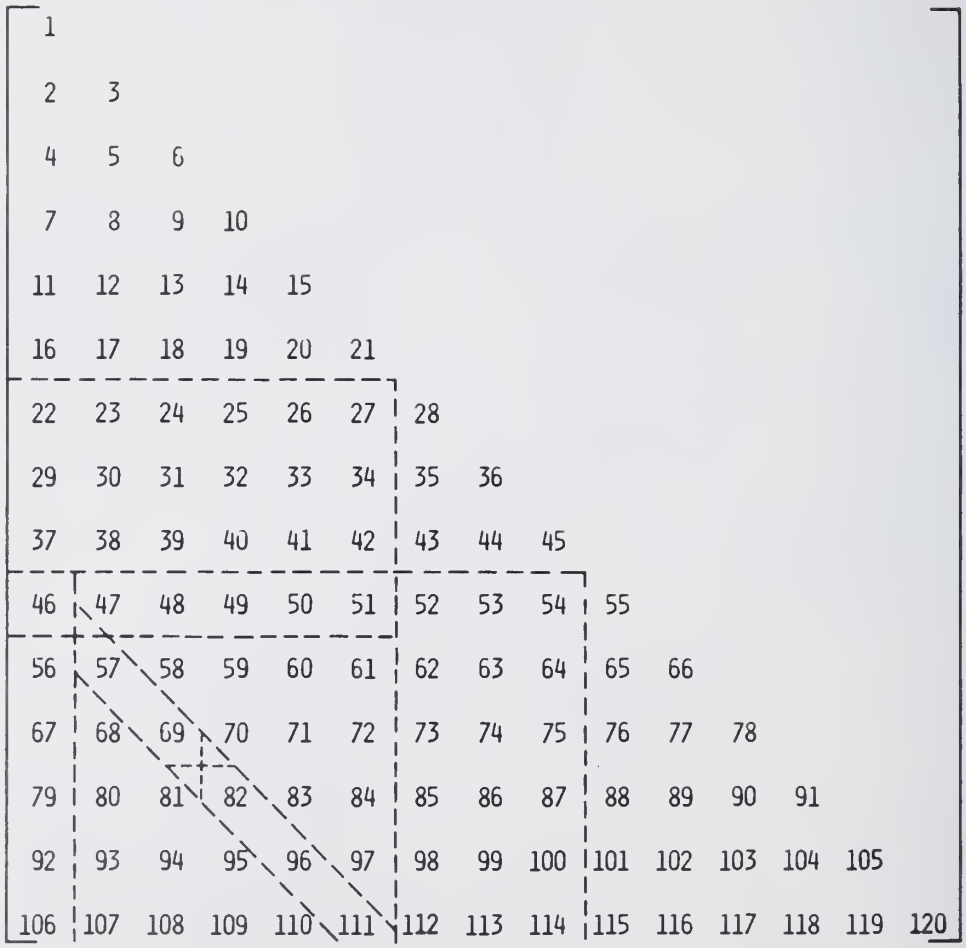

FIG. 6. Multilevel partitioning of a lower triangular array and relative addresses of the elements.

single ACB node for all A<I,J>, but we have not chosen this alternative.

Figure 5 illustrates the tree structure corresponding to Figure 2 using simplified ACB nodes and partition control blocks. Most of the information displayed in Figure 5 follows from Figure 6 and from the example which defined Figure 2. A more complete description, including a derivation of the fields not shown, is discussed in the next section.

Basic Algorithms

The purpose of this section is to define when an ACB node should be added to a tree structure and how the information for a new node is to be derived. We define these processes in terms of basic algorithms.

In order to interpret the examples presented in this section and in the appendix, we define the following semantic rules which, in general, are not included in the basic algorithms.

1. LET statements declare arrays and therefore are non-executable. Within a block all LET statements are processed before any executable statement.

2. PARTITION and SET statements are executable.

3. Once a PARTITION statement is executed, any partitioned array in that statement remains partitioned until control passes out of the block, or until another partition statement in the same block redefines the partitioning for that array (an exception is explained in 5). If a new partitioning is defined for that same array, then the subtree for the previous partition is deleted and a new subtree is built using the basic algorithms.

If the same partition statement is executed more than once and no other partition statement redefines the partitioning for that same array, then the subtree remains intact and the fields of the specified ACB or PCB are merely updated.

4. Once a SET statement is executed such as, "SET X=A<2,1>," the subarray name X may be used until control passes out of the block, or until another SET statement uses the same name for a different subarray (an exception is explained in 5). If the name X is "switched" to another subarray, then the previous subarray A<2,1> cannot be referenced, but its ACB node is not destroyed. This allows the same name X to be used for different subarrays without the overhead associated with creating and destroying nodes and subtrees. If the same SET statement is executed more than once and no other SET statement uses the same name, then subsequent executions will update the subtree node for X. However, if the subarray is A<I,J>, then a new node will be created whenever the values of the subscripts <I,J> reference a new subarray A<I,J>. In this case the name X always refers to the last subarray referenced. Thus, a single SET statement which is executed repetitively can generate many subtree nodes, one for each distinct pair of values of <I,J>.

5. A SET statement of the form SET A=B, where B is defined by some previous statement, is treated in a different manner than described above. In particular, an ACB node is created for A, which is an exact copy of the ACB node for B. Then, the ACB node for A becomes a root node of a new tree structure, its "descendent tree," which is independent of the tree structure which contains

the ACB node for B. If another SET statement redefines A, then the existing ACB node for A and its subtree nodes are deleted and a new node is created using the basic algorithms. On the other hand, if the same SET statement is executed more than once, then the ACB for A is updated, provided A has not been used in another SET statement in which case an ACB node for A must be recreated using the basic algorithms.

The notation which is used in the following algorithms is similar to that found in Knuth [6], however, at times it is necessary to introduce additional notation. This is explained either within the algorithm or in the discussion which follows it.

ALGORITHM R (Root Node) The root node of a tree structure is constructed in this algorithm. It is a straightforward initialization of the root node fields from information contained in a LET statement as discussed in previous sections.

We denote by A the name of the array and by $\vec{A}$ the explicit pointer variable that is used to reference the ACB which describes A.

R1. [Initialize] Allocate ACB, set R. The size of the ACB is actually a function of the dimension of the array, which is available at compile time. Once allocated the pointer R is set to point to this ACB node, which is now designated by node(R). Similarly, the fields of node(R) are designated by $\eta(R)$, $\Delta(R)$, $\tau(R)$, ..., $\rho(R)$, where the symbols are defined in Table II. Allocate A, set $\omega(R)$. This allocates memory for the array elements; the array bounds and the array type determine the amount of memory required. $\omega(R)$ is the location of the first stored element of the array (for the two dimensional arrays of Table I this corresponds to either the (1,1), (1,2), or (2,1) element).

R2. [Fill ACB] $\eta(R) \leftarrow$ name of array, $\tau(R) \leftarrow$ type of array (default is rectangular), $\Delta(R) \leftarrow$ dimension of array, $\lambda_i(R) \leftarrow 1$ lower bound for i-th dimension (defined to be unity for each dimension), $\mu_i(R) \leftarrow$ upper bound for i-th dimension, $\rho(R) \leftarrow \Lambda$ (null pointer). Set $\delta R(R)$ and $\delta D(R)$ as defined in Table III.

R3. [Set ACB pointer] $\vec{A} \leftarrow R$, terminate algorithm.

If other arrays are declared in the same LET statement or if there are other LET statements in the same block, then this algorithm is repeated until all root nodes are filled. Once this is completed the remaining algorithms can be used to generate subtree nodes for the tree structures as needed.

As an example consider the LET statement which generates the root node in Figures 2 and 5, namely

LET A BE A LOWER TRIANGULAR MATRIX OR ORDER (15).

Using the relative addressing indicated in Figure 6 and applying algorithm R, we obtain the first line of Table V. The other entries in Table V are the result of applying subsequent algorithms and are discussed at the appropriate place.

ALGORITHM P (Partition Array) Let P be a pointer to an ACB node in a tree structure, and let a PARTITION statement be defined for the array described by node(P). Denote by R_I ($0 \le I \le N$) and C_J ($0 \le J \le M$) the row and column expressions in the PARTITION statement, and assume that either $N \neq 0$ or $M \neq 0$ so that we have at least one partition line. Then, this algorithm fills the PCB with the necessary partitioning information. If the PCB doesn't exist, it is created in step P1. Q is a pointer which is used to reference the PCB.

P1. [Locate PCB] If $\tau(P) \neq R$ then $N \leftarrow \max(N,M)$, $M \leftarrow N$. Set $Q \leftarrow \rho(P)$. If $Q \neq \Lambda$ then go to step P2; otherwise allocate PCB, set Q, $\rho(P) \leftarrow Q$, $\pi(Q) \leftarrow \Lambda$, and $\sigma(Q) \leftarrow \Lambda$. (The number of pointers in $\sigma(Q)$ depends on the type $\tau(P)$ and the number of partition lines, namely M+N). Set $\nu(Q) = M+N$.

P2. [Repartition?] If $\nu(Q) = M+N$ then go to step P3; otherwise delete subtrees of node(P) which are pointed to by $\sigma(Q)$. Save $\pi(Q)$, reallocate other parts of PCB, $\nu(Q) \leftarrow M+N$, $\sigma(Q) \leftarrow \Lambda$. (The size of the PCB is adjusted when repartitioning of an array occurs but $\pi(Q)$ is left unaltered.)

P3. [Fill PCB] If $\tau(P)$ is not rectangular in step P1, either N or M is zero, since either R_I or C_J is specified but not both. In such a case, interpret $R_K = C_K$ for $1 \le K \le \max(M,N)$ in the following assignments. Set $r(Q,0) \leftarrow \lambda_1(P) - 1$, $r(Q,I) \leftarrow R_I$ $1 \le I \le N$, $r(Q,N+1) \leftarrow \mu_1(P)$. Similarly, $c(Q,0) \leftarrow \lambda_2(P) - 1$, $c(Q,J) \leftarrow C_J$ $1 \le J \le M$, $c(Q,M+1) \leftarrow \mu_2(P)$.

P4. [Order partitions] Sort r(Q,I) I≠0, I≠N+1, and c(Q,J) J≠0, J≠M+1 into ascending order. At execution time, the partition lines are ordered according to the values of the expressions R_I and C_J and not according to the subscripts I and J. Terminate algorithm.

If other arrays are to be partitioned with the same partition statement, then this algorithm is repeated the necessary number of times. In each case we assume the existence of an algorithm which, if necessary, traverses the tree at execution time to find the correct ACB node and set the pointer P. For instance, if A is the name of the array to be partitioned, then a tree traversal is not necessary because the explicit pointer $\vec{A}$ exists and P←$\vec{A}$ locates the correct ACB node. On the other hand, if A<2,1> is the array to be partitioned, then no explicit pointer exists for the ACB node for A<2,1>. Therefore, we traverse a subtree by starting with $\vec{A}$, then locating the PCB through $\rho(\vec{A})$, and finally locating the ACB for A<2,1> by using σ(Q,2,1) which is the unique pointer field in σ(Q) that is reserved for the ACB for A<2,1>. The pointer Q is, of course, equal to $\rho(\vec{A})$. Thus, in this case P would assume the value of σ(Q,2,1). This general procedure can be extended to more than two levels. Thus, whenever an algorithm assumes that P points to an ACB node, we will assume that its value was obtained by a simple traversal algorithm of this type.

As indicated, we use σ(Q,I,J) to refer to the pointer in σ(Q) which points to the ACB node for A<I,J>, where A is the array that is partitioned. For our purposes we may think of σ(Q), as a two dimensional array of pointers, one for each possible subarray. However, in practice the size of σ(Q) is reduced to the maximum

number of subarrays that can legally be referenced. For instance if A is lower triangular then there are only (N+1)(N+2)/2 subarrays defined by N row partitions R_1, ..., R_N.

Consider now the situation where another partition statement repartitions the same array. The old partition must be deleted, along with all of the subtrees, and the size of the PCB must be adjusted to account for a different number of subarrays. This is accomplished in step P2.

If τ is lower triangular, we denote by $\pi(Q,\tau)$ the pointer in $\pi(Q)$ which points to the ACB node which describes the lower triangular part of A. Thus in step P1, $\pi(Q)\leftarrow\Lambda$ simply initializes all the pointers in $\pi(Q)$. The pointer fields in $\pi(Q)$, therefore are available later for diagonal partitioning (Algorithm D) if needed.

Algorithm P may create a PCB; however, if diagonal partitioning is encountered first, then Algorithm D creates the PCB. In this case $\sigma(Q)$ is empty, $(\nu(Q)=0)$, thus if a partitioning statement is encountered and a PCB exists, it becomes necessary to expand the PCB in step P2 without altering the pointers in $\pi(Q)$.

Rectangular arrays may be partitioned after rows and columns, or after rows only or after columns only. In the case of vectors, only the latter are meaningful. In such cases, we define A<I,0>=A<I> if M=0 and A<0,J>=A<J> if N=0. It is also obvious that in such cases $\sigma(Q,I,0)=\sigma(Q,I)$ becomes a simple array of pointers of length N+1 or M+1. Partitions of this type occur in the appendix, and the algorithms defined here apply using the above notational changes.

To illustrate Algorithm P, consider the statement

PARTITION A AFTER ROWS 6,9

which corresponds to the PCB at level 1 in Figure 5. The PCB is created in step P1 and filled in step P3. Since $\tau(P)$ is lower triangular the quantities C_1 and C_2 in step P3 are determined from R_1 and R_2. That is, by definition, the partitioning lines $R_1=6$ and $R_2=9$ are reflected off of the main diagonal giving $C_1=6$ and $C_2=9$. It is also important to note that the pointers in σ and π are set to Λ in step P1, and it is the responsibility of subsequent algorithms, namely S and D, to insert pointers into the appropriate places in σ and π to create the branches shown in Figure 5.

ALGORITHM S (Subtree Node) Let P be a pointer to an ACB node, and let a SET statement be defined which sets an array, say C, to the subarray described by node(P). For example, "SET C = A<2,1>." Then, this algorithm defines the rules for constructing a subtree node for C. We assume that P is determined as in Algorithm P. Throughout this algorithm, P points to the parent node, Q to the PCB for node(P), and S to the ACB subtree node. The pointer P is obtained by a simple traversal of the tree if necessary, but for the above example it would be $P \leftarrow \vec{A}$.

S1. [Initialize] Set $Q \leftarrow \rho(P)$. We assume that a PCB has been filled by algorithm P; otherwise, the partitioning is undefined.

S2. [ACB exist?] $S \leftarrow \sigma(Q,I,J)$ where I and J are extracted from the subarray subscript pair <I,J> in a reference of the form A<I,J>. If $S=\Lambda$ then allocate ACB, set S, $\sigma(Q,I,J) \leftarrow S$, $\rho(S) \leftarrow \Lambda$.

S3. [Fill ACB] Set $\delta D(S) \leftarrow \delta D(P)$. If I=J then $\tau(S) \leftarrow \tau(P)$, otherwise $\tau(S) \leftarrow R$. Set $\lambda_i(S) \leftarrow 1$ for i=1,2. $\mu_1(S) \leftarrow r(Q,I) - r(Q,I-1)$, $\mu_2(S) \leftarrow c(Q,J) - c(Q,J-1)$, $\delta R(S) \leftarrow \delta R(P) + \delta D(P)\ r(Q,I)$. If $\tau(P)$=SLT then $\delta i \leftarrow r(Q,I-1)-1$, $\delta r \leftarrow \delta R(P) + \delta D(P)$; otherwise $\delta i \leftarrow r(Q,I-1)$, $\delta r \leftarrow \delta R(P)$. If $\tau(P)$=SUT then $\delta j \leftarrow c(Q,J-1)-1$; otherwise $\delta j \leftarrow c(Q,J-1)$. Set $\omega(S) \leftarrow \omega(P) + \delta r\ \delta i + \delta D(P)(\delta i(\delta i-1))/2 + \delta j$. If the attribute "VECTOR" follows the subarray reference then set $\Delta(S) \leftarrow 1$ and go to step S4. If the attribute "SCALAR" follows the reference then set $\Delta(S) \leftarrow 0$, and go to step S4; otherwise set $\Delta(S) \leftarrow \Delta(P)$.

If $\tau(S)$=SLT then $\omega(S) \leftarrow \omega(S) + \delta R(S)$. If $\tau(S)$=SUT then $\omega(S) \leftarrow \omega(S) + 1$.

S4. [Name node] Set $\vec{C} \leftarrow S$, $\eta(S) \leftarrow$ 'C', where C is the name of the subarray which appears in the SET statement.

The first time a SET statement is executed an ACB is created for the subtree node. Subsequent executions of the same SET statement may or may not create another ACB, it depends upon whether I and J have values which are distinct from the previous execution. However, in many applications it is sufficient to use constants in place of I and J, then the ACB node is merely updated on subsequent executions. Examples which illustrate this point are presented in the Appendix.

We note that the pointer $\vec{C}$ is always set in step S4 to point to the current ACB node for the subarray A<I,J>. Within a loop, therefore, C may be used to refer to different subarrays of the form A<I,J>. The generality of this feature includes even the possibility of varying the type of the subarray. For example, if A is lower triangular, then A<I,J> is lower triangular if I=J and rectangular otherwise, so that the type is clearly a function of the subscript values.

The values of δi and δj in step S3 are obtained from the PCB fields r(Q,I-1) and c(Q,J-1), which represent the current position of the partitioning lines. Also, recall that Algorithm P insures that the partitioning lines are in ascending order; it follows, therefore, that $\mu_i(S) \geq 0$ in step S3. If $\mu_i(S)=0$ for some i, then we define the subarray which has both $\lambda_i=0$ and $\mu_i=0$ to be a "NULL" array. This is discussed in the next section. The case where r(Q,J) assumes a negative value is not allowed.

We can illustrate algorithm S by considering the statement

SET B=A<2,1>, C=A<2,2>, D=A<3,2>

which builds the ACB nodes for B, C, and D in Figure 5 at level 2. The values of the fields are shown in Table V. Each node requires a separate application of Algorithm S.

The attributes VECTOR and SCALAR in step S3 play a special role. In essence, they allow the compiler to invoke a precedence relation when compiling array expressions so that the number of operations can be reduced. The essential point is that the attribute characterizes the shape of the dynamic subarray as it moves over its parent array, and the attribute guarantees that the shape will not change. The examples in the appendix show that array algorithms can be constructed which have dynamic subarrays of constant shape. Though array evaluation is not central to partitioning, it is vital to array language design and deserves some mention at this point.

ALGORITHM D (Diagonal Partitioning) Let P be a pointer to an ACB node, and let a SET statement be defined which sets an array, say B, to a "part of" the array described by node(P). For example, "SET B TO THE LOWER TRIANGULAR PART OF . . ." is typical. This algorithm prescribes the rules for deriving the information for the subtree node from the parent node and the given SET statement.

D1. [Locate PCB] Set $Q \leftarrow \rho(P)$. If $Q \neq \Lambda$ then go to step D2, otherwise allocate PCB, set Q, $\rho(P) \leftarrow Q$, $\pi(Q,\tau) \leftarrow \Lambda$ for all τ. Set $\nu(Q)=0$ (indicates that $\sigma(Q)$, $r(Q,I)$, and $c(Q,J)$ are empty.)

D2. [ACB exist?] Set $S \leftarrow \pi(Q,\tau)$ where τ is the type prescribed in the SET statement. If $S \neq \Lambda$ then go to step D3, otherwise allocate ACB, set S, $\pi(Q, \tau) \leftarrow S$, $\rho(S) \leftarrow \Lambda$.

D3. [Fill ACB] Set $\Delta(S) \leftarrow \Delta(P)$, $\delta D(S) \leftarrow \delta D(P)$, $\delta R(S) \leftarrow \delta R(P)$, $\lambda_i(S) \leftarrow 1$ i=1,2, $\tau(S) \leftarrow$ type of subarray prescribed in SET statement. Set $\mu_i(S)$, and $\omega(S)$ according to Table IV.

D4. [Name node] Set $\eta(S) \leftarrow$ 'B' where B is the name of the subarray given in the SET statement. Set $\vec{B} \leftarrow S$ where $\vec{B}$ is the pointer variable which corresponds to the name B. Terminate algorithm.

TABLE IV Rules for Calculating $\mu_i(S)$ and $\omega(S)$ in Algorithm D

$\tau(S)$ \ $\tau(P)$	SLT	LT	D	SUT	UT	TD	R
SLT	X	2,4	X	X	X	X	1,4
LT	X	X	X	X	X	X	1,5
D	X	2,5	X	X	2,5	2,5	1,5
SUT	X	X	X	X	2,3	X	1,3
UT	X	X	X	X	X	X	1,5
TD	X	X	X	X	X	X	1,5
R	X	X	X	X	X	X	X

Key	Rule	
1	$\mu_i(S) \leftarrow \min(\mu_1(P),\mu_2(P))$	i=1,2
2	$\mu_i(S) \leftarrow \mu_i(P)$	i=1,2
3	$\omega(S) \leftarrow \omega(P) + 1$	
4	$\omega(S) \leftarrow \omega(P) + \delta R(P)$	
5	$\omega(S) \leftarrow \omega(P)$	
X	Not Allowed	

The condition that P points to the parent node in Algorithm D implies the existence of a traversal algorithm. For instance, consider the statement

SET U TO THE STRICTLY UPPER TRIANGULAR PART OF A<3,1>

which generates two ACB nodes in Figure 5, namely the node for A<3,1> and the node for U. In order to obtain P, we assume that the traversal algorithm starts with the explicit pointer $\vec{A}$, finds the PCB through $Q \leftarrow \rho(\vec{A})$, and discovers that $\delta(Q,3,1)$, the pointer for A<3,1> is Λ. Thus, it invokes algorithm S to create and fill an ACB node for A<3,1>; only then does it have the correct pointer for P. Algorithm D can now be used to create the ACB node for U, and incidentally a PCB as well. Table V shows the values of the fields for these two ACB nodes.

Another feature of the above example is that the PCB which is created in step D1 consists of only π. When the next statement, namely

PARTITION A<3,1> AFTER ROW 1 AND COLUMN 1

is encountered, Algorithm P must adjust the size of the PCB to accomodate the row and column partitions. Once this is done, the PCB takes the form as shown in Figure 5.

Algorithm D is also applied in creating the ACB nodes for A<3,1><2,2>, L, and E in Figure 5 and Table V. The statement which invokes Algorithm D is

SET L TO THE LOWER TRIANGULAR PART OF A<3,1><2,2>,
E TO THE DIAGONAL PART OF A<3,1><2,2>.

The interpretation is the same as described previously.

Table IV not only describes the rules for determining the upper bounds μ_i and the origin ω as a function of the array type, but also specifies the types of subarrays that are allowed. According to our previous definition, the subarray may be of the same type as the parent array. Table IV does not allow this. The reason is that under diagonal partitioning you would be merely duplicating the parent ACB node. Instead, we create a new root node using Algorithm E so that we may have an independent description of any array or subarray. The new root nodes are different from other root nodes because they are not created by Algorithm R. The new root nodes in a sense, give rise to descendent trees because they inherit their original description from some tree node; however, once they are created they may grow independently of their parent tree. In this way we obtain independent partitions which have also been referred to as overlaying partitions.

TABLE V Values of ACB Fields for Figure 5 Using Basic Algorithms

ALGORITHM	ARRAY OR PARENT	NAME FIELD	TYPE FIELDS		CONTROL FIELDS			VIRTUAL BOUNDS			
		η	Δ	τ	ω	δR	δD	λ_1	μ_1	λ_2	μ_2
R	A	'A'	2	LT	1	1	1	1	15	1	15
S	A<2,1>	'B'	2	R	22	7	1	1	3	1	6
S	A<2,2>	'C'	2	LT	28	7	1	1	3	1	3
S	A<3,2>	'D'	2	R	52	10	1	1	6	1	3
S,D	A<3,1>	empty	2	R	46	10	1	1	6	1	6
D	A<3,1>	'U'	2	SUT	47	10	1	1	6	1	6
S,D	A<3,1><2,2>	empty	2	R	57	11	1	1	5	1	5
D	A<3,1><2,2>	'L'	2	LT	57	11	1	1	5	1	5
D	A<3,1><2,2>	'E'	2	D	57	11	1	1	5	1	5
S	E<2,2>	'F'	2	D	82	13	1	1	3	1	3

ALGORITHM E (Equivalent node) Suppose a SET statement of the form "SET A=B" is given, where B denotes any array (or subarray) which has been defined previously. We assume that B is not qualified, that is, there are no attributes or subarray subscripts attached to B. Secondly, we assume that the explicit pointer $\vec{A}$ is either Λ, which implies that A has not been used; or that $\vec{A}$ points to some subtree node which implies that it is to be redefined by this algorithm, or that $\vec{A}$ points to a marked node which implies that the node was previously created by this algorithm. We have not previously indicated the existence of a marking bit in the ACB node, but we assume that it now exists. This algorithm duplicates the information of node($\vec{B}$) in node($\vec{A}$).

E1. [ACB exist?] Set $E \leftarrow \vec{A}$. If $E=\Lambda$ go to step E2. If node(E) is marked go to step E3; otherwise go to step E2. (Any node which is marked must have been created by this algorithm and must be a root node of a descendent tree.)

E2. [Create ACB] Allocate ACB, set E, $\rho(E) \leftarrow \Lambda$, mark node(E).

E3. [Copy Fields] Set $\tau(E) \leftarrow \tau(\vec{B})$, $\delta R(E) \leftarrow \delta R(\vec{B})$, $\delta D(E) \leftarrow \delta D(\vec{B})$, $\lambda_i(D) \leftarrow \lambda_i(\vec{B})$, $\mu_i(E) \leftarrow \mu_i(\vec{B})$, $\Delta(E) \leftarrow \Delta(\vec{B})$.

E4. [Name node] Set $\eta(E) \leftarrow$ 'A', $\vec{A} \leftarrow E$ and terminate algorithm.

If the SET statement which invokes this algorithm is executed more than once, then subsequent executions will merely update the ACB node for A with the current information in node($\vec{B}$). There are no restrictions imposed on B; it may be an arbitrary array as long as it is defined.

After Algorithm E is executed for some SET statement, the array A may be partitioned, and any such partitioning is independent of any partitioning for B. This idea may be extended to include any number of independent partitions of B by simply adding more SET statements of the form described.

It is also possible to execute the SET statement, and then change B either through another SET statement or partition statement. If this is done then it is clear that A continues to describe the original B and will continue to do so as long as Algorithm E is not invoked a second time. The first example in the appendix illustrates an application of Algorithm E by using the statement SET L = A.

Null Operands

Recall that, in general, the position of a partitioning line is defined by a value of an expression in a partitioning statement. This implies that references to the subarrays cannot be determined until execution time; it also implies that some of the partitioning lines may be positioned outside of the array boundaries, or that several partitioning lines may be superimposed. Each of these problems must be solved in order to uniquely define the value of array expressions.

The first problem is solved by ordering the contents of the r_i and c_j fields in the PCB into ascending order during the execution of the partition statement. This uniquely defines the boundaries of each subarray, provided all of the partitioning lines which define the subarrays are distinct and lie on the boundary or inside of the parent array.

If an array is partitioned and if a partitioning line lies outside of the array boundaries, then all subarrays which are bordered by that line are defined to have a null value. Similarly, if two partitioning lines are superimposed, then all subarrays which would normally lie between these two lines, if they were not superimposed, are also defined to have a null value. In an array expression, an operand which has a null value is defined to be a null operand.

In the OL/2 language, a null operand in an array expression obeys the same rules as the additive identity. That is, the null operand behaves like a zero matrix, a zero vector, or a zero scalar, except in scalar division, in which case it is equivalent to no-operation. The main differences between a null operand and a zero operand is that the zero operand requires actual computation while the null operand is combined with other operands without any computation.

One of the interesting by-products of defining null operands is that it simplifies the writing of array algorithms. In general, the simplification occurs in the initial and final steps of an algorithm, which normally require special consideration, but with null operands are automatically included in the general case. The examples in the appendix illustrate this point.

Conclusion

The purpose of this paper was to introduce a general partitioning strategy for array languages. By defining different types of arrays and various modes of partitioning we were able to obtain an efficient and powerful means of accessing subarrays. Since partitioning is a fundamental operation that is required in implementing many array algorithms, it is of general interest to the user and the designer of array languages.

Of prime interest to the language designer are the basic algorithms and the design of the information structure which describes dynamic subarrays of various shapes and sizes. Whenever possible, we have tried to make the basic algorithms independent of the semantics associated with OL/2, so that they may be applied to the design of other array languages. It is evident that other basic types of arrays could be added to those we have considered; for instance, triangular arrays which are obtained by rotating either lower or upper triangular arrays by 90 degrees. However, one should not that the basic array types which we have described are adequate for handling the standard types of sparse matrices which arise in applications of partial differential equations, namely, band matrices and block tridiagonal matrices. The technique is to realize that these types of sparse matrices can be constructed by taking combinations of the basic types of arrays. This, of course, requires one to introduce additional data types, but the underlying information structure remains the same.

For the user, partitioning provides a vital and convenient means for accessing parts of arrays. The result is that one may now apply an array language to the problem of constructing array algorithms, not only for iterative methods, but also for direct methods. For properly written array algorithms, the number of operations for any of the iterative or direct methods is the same as for an element language, and the amount of memory space is always optimal for each array. Only in the evaluation of array expressions is additional memory space required, and this can always be automatically generated by the compiler.

APPENDIX

The examples given here are written in the OL/2 language, the purpose being to illustrate the practical aspects of partitioning. In both examples the number of operations and the amount of storage is nearly optimal, the difference being that one temporary vector is needed in the evaluation of some of the array expressions.

There are several variations of the syntax, some more compact than others. Here we present a natural and descriptive form which requires relatively little explanation, yet is acceptable to the compiler.

```
CHOLESKY_METHOD:  PROCEDURE(A,X,Z,N);
    LET  A  BE A LOWER TRIANGULAR MATRIX OF
ORDER (N);   LET  X,  Y,  AND  Z  BE VECTORS OF
ORDER (N);   SET L=A;
    FOR K=1,2,...,N;   PARTITION  A,  Y,  AND  Z
AFTER ROWS K-1, K;   SET R=A<2,1> ROW VECTOR,
D=A<2,2> SCALAR, M=A<3,1>, C=A<3,2>  COLUMN VECTOR,
U=Y<2>, V=Z<2>, AND W=Y<1>;
            D = SQRT( D - (R',R') );
            C = ( C - M×R' )/D;
            U = ( V - R×W )/D;
END;
BACK_SUBSTITUTION:
    FOR K=N, N-1,...,1;   PARTITION  L,  X,  AND
Y  AFTER ROWS K-1,  K;   SET C=L<3,2>, D=L<2,2>
SCALAR, S=X<2>, U=Y<2>, AND T=X<3>;
            S = ( U - C'×T )/D;
END CHOLESKY_METHOD;
```

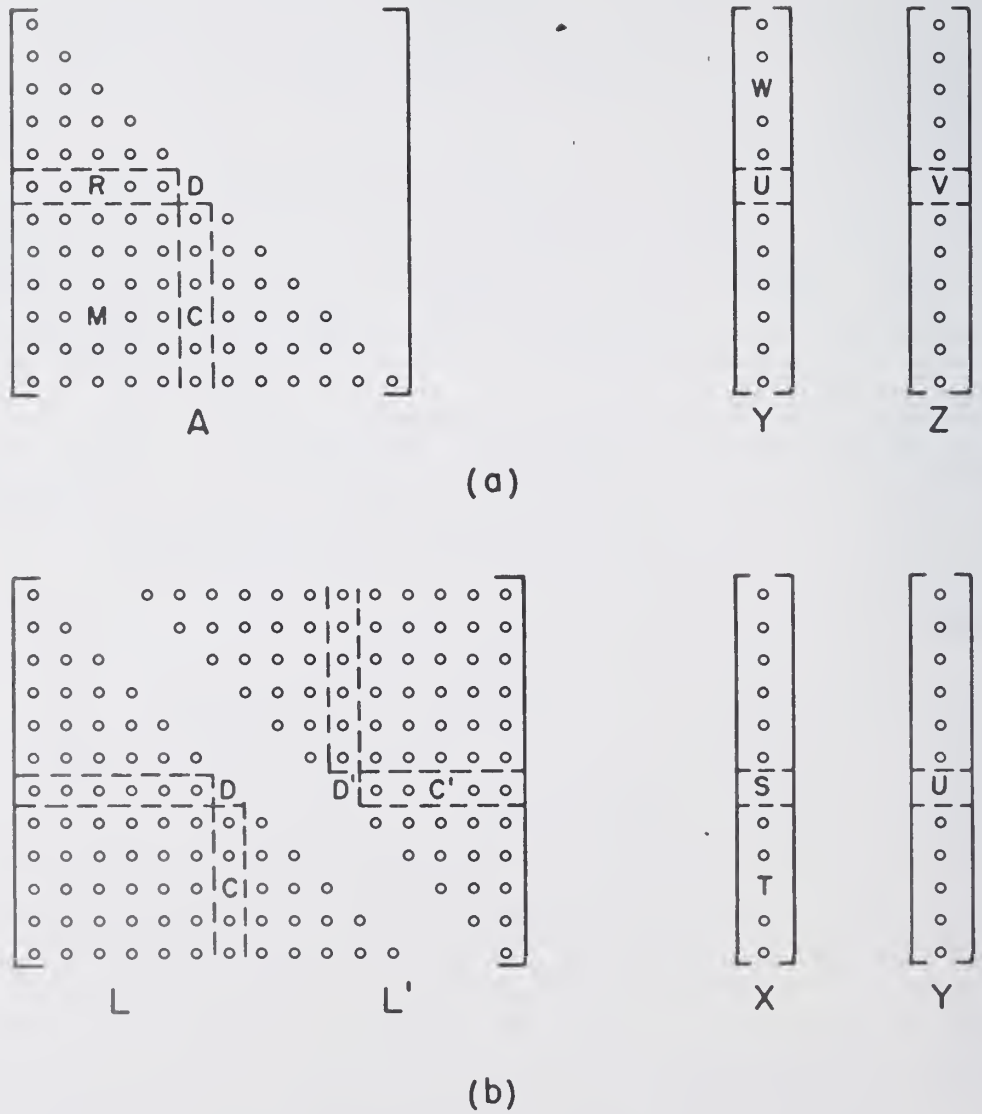

FIG. 7. Cholesky method: (a) decomposition and forward substitution; (b) back substitution.

The Cholesky method for solving a system Ax=z is preferred over other direct methods when A is dense, symmetric, and positive definite. Since A is symmetric it can be stored as a lower triangular matrix. The first part of the algorithm decomposes the matrix A into the product L·L' and stores L into A without using additional storage. The forward substitution is carried out at the same time so that y contains the solution of Ly=z. Finally, the back substitution solves L'x=y for x. Since A is positive definite, division by zero is not possible; however, an ON condition, as in PL/1, may be inserted if desired. Finally, observe that A and L refer to the same array, but they are partitioned independently of each other.

The prime denotes the transpose of a vector or matrix. Vectors which are declared in LET statements are column vectors. The END statement acts as a delimiter for procedure blocks and FOR statements as in PL/1, and it may also serve as a RETURN as in PL/1. The arithmetic assumes the default attributes of double precision floating point except in the case of the undeclared variables K and N which are integers.

All array operations follow the usual rules of linear algebra, except when an array becomes a null operand. This usually occurs in the initial and final steps of an algorithm when the partitioning lines force some of the subarrays to lie outside of the original array. For instance, when K=1 in the first example, the subarrays R, M, and W in Figure 7 become null operands, and the array expressions degenerate into D = SQRT(D), C = (C)/D, and U = (V)/D respectively. Formally,

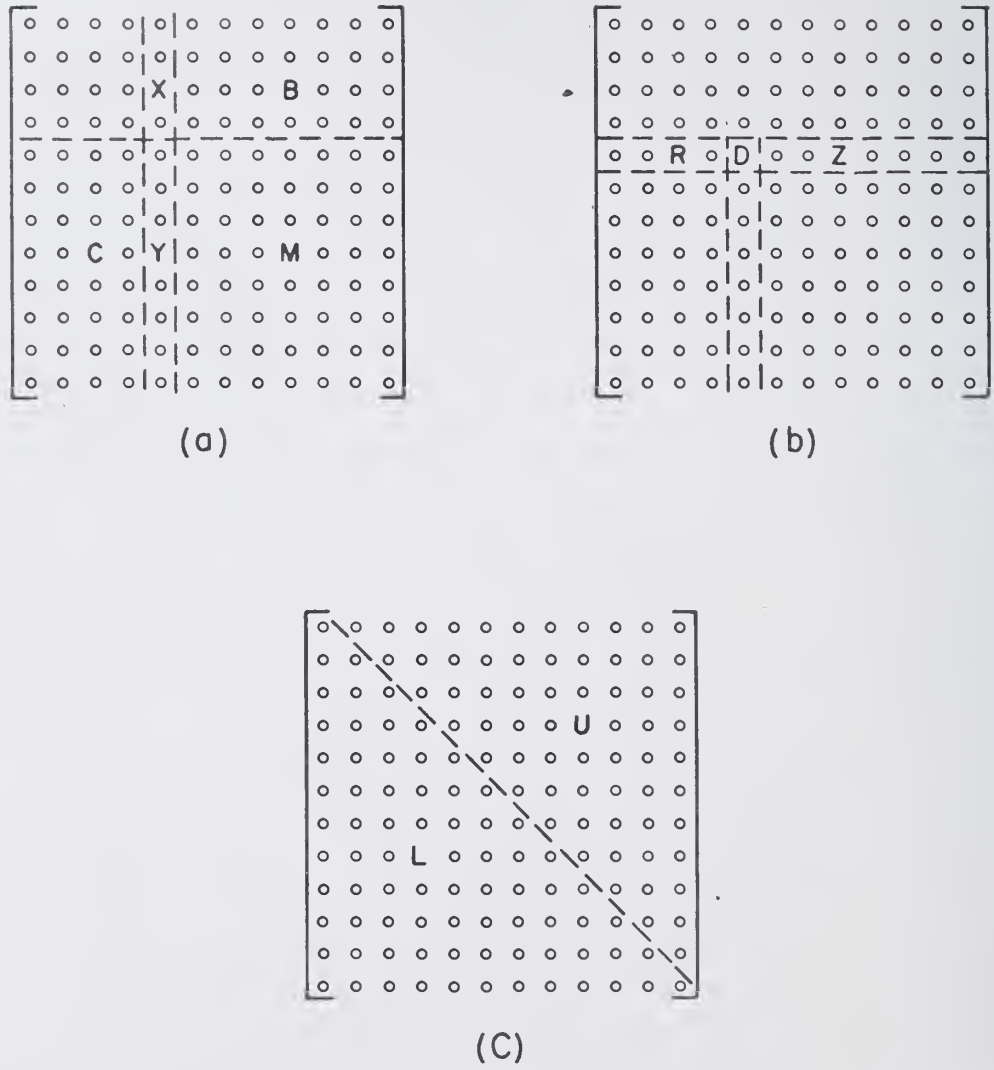

FIG. 8. Crout decomposition: (a) first level rectangular partitioning of A; (b) second level rectangular partitioning of C, Y, and M; (c) first level diagonal partitioning of A.

null operands obey the same rules as the additive identity. Since there is an additive identity for vectors, matrices, and scalars the rules are well defined. The only exception is division by a null operand which produces a null result rather than an undefined operation. Clearly, null operands can be handled without actually carrying out the computations.

The second example is the familiar Crout algorithm. This algorithm requires two levels of partitioning to decompose a matrix A into the product L·U. Theoretically, L is lower triangular and U is unit upper triangular, but in practice the diagonal elements of U are not stored. Therefore, U is declared as a strictly upper triangular matrix. As the algorithm progresses, the elements of L and U are formed and stored in A. Figure 8 illustrates the process for some intermediate value of K.

```
LU_DECOMPOSITION:  PROCEDURE(A,N);
     LET  A  BE A MATRIX OF ORDER (N);   SET L TO
THE LOWER TRIANGULAR PART OF  A,  AND  U  TO THE
STRICTLY UPPER TRIANGULAR PART OF  A;
     FOR K=1,2,...,N;  PARTITION  A  AFTER ROW K-1
AND AFTER COLUMNS K-1,  K;   SET C=A<2,1>, X=A<1,2>,
Y=A<2,2>, B=A<1,3>, AND M=A<2,3>;   PARTITION C, Y, M
AFTER ROW 1;  SET R=C<1>, D=Y<1> SCALAR, AND Z=M<1>;
                    Y = Y - C×X;
                    Z = ( Z - R×B )/D;
END LU_DECOMPOSITION;
```

Though L and U are specified, they are not used in this example except to clarify the algorithm. However, another segment could be added which would use L and U to solve one or more systems of equations $Ax_i=z_i$ in the usual way. Finally, partial pivoting may be introduced by using a slightly different partitioning and an interchange statement; however, this does not add anything new to the example.

REFERENCES

1. ADAMS, H. C. Dynamic partitioning in the array language OL/2. DCL Report No. 421, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana, Illinois, Jan. 1971.

2. BAYER, R., AND WITZGALL, C. Some complete calculi for matrices. Comm. ACM 13, 4 (April 1970), pp. 223-237.

3. COHEN, S., AND VINCENT, C. M. An introduction to SPEAKEASY. Report PHY-1968E, Argonne National Laboratory, Argonne, Illinois, Dec. 1968.

4. DANTZIG, G. B., ET AL. MPL mathematical programming language specification manual for committee review. STAN-CS-70-187, Computer Science Dept., Stanford Univ., Stanford Calif., Nov. 1970.

5. IVERSON, K. E. A Programming Language. Wiley, New York, London, 1962.

6. KNUTH, D.E. The Art of Computer Programming. Fundamental Algorithms, Vol. 1, Addison Wesley, Reading, Mass., 1969.

7. PHILLIPS, J. R., AND ADAMS, H. C. Dynamic partitioning for array languages. To appear in CACM 1972.

BLIOGRAPHIC DATA EET	1. Report No. UIUCDCS-R-71-420	2.	3. Recipient's Accession No.
Title and Subtitle: THE STRUCTURE AND DESIGN PHILOSOPHY OF OL/2--AN ARRAY LANGUAGE - PART II: Algorithms for dynamic partitioning			5. Report Date: September, 1971 6.
Author(s): J. RICHARD PHILLIPS			8. Performing Organization Rept. No.
Performing Organization Name and Address: Dept. of Computer Science University of Illinois at Urbana-Champaign Urbana, Illinois 61801			10. Project/Task/Work Unit No. 11. Contract/Grant No.: US NSF-GJ-328
Sponsoring Organization Name and Address: National Science Foundation Washington, D.C. 20550			13. Type of Report & Period Covered 14.

Supplementary Notes

Abstracts

The classical process of partitioning an array into subarrays is extended to a more useful array language operation. Various modes of partitioning are defined for different types of arrays, so that subarrays may vary over the original array in a nearly arbitrary manner. These definitions are motivated with several realistic examples to illustrate the value of partitioning for array languages.

Of general interest is the data structure for partitioning. This consists of dynamic tree structures which are used to derive and maintain the array control information. These are described in sufficient detail to be of value in the design of other array languages. The description presented in this paper is implemented in a new array language, OL/2, currently under development at the University of Illinois.

Key Words and Document Analysis. 17a. Descriptors

Dynamic partitioning, array partitioning, array language, data structure, tree structure, programming language design, array control blocks, partition control blocks.

b. Identifiers/Open-Ended Terms

c. COSATI Field/Group

Availability Statement	19. Security Class (This Report)	21. No. of Pages
Unlimited	UNCLASSIFIED	57
	20. Security Class (This Page) UNCLASSIFIED	22. Price